Capítulo 5

PROGRAMACIÓN DINÁMICA

5.1 INTRODUCCIÓN

Existe una serie de problemas cuyas soluciones pueden ser expresadas recursivamente en términos matemáticos, y posiblemente la manera más natural de resolverlos es mediante un algoritmo recursivo. Sin embargo, el tiempo de ejecución de la solución recursiva, normalmente de orden exponencial y por tanto impracticable, puede mejorarse substancialmente mediante la Programación Dinámica.

En el diseño Divide y Vencerás del capítulo 3 veíamos cómo para resolver un problema lo dividíamos en subproblemas independientes, los cuales se resolvían de manera recursiva para combinar finalmente las soluciones y así resolver el problema original. El inconveniente se presenta cuando los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos; entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que conlleva. En estos casos es cuando la Programación Dinámica nos puede ofrecer una solución aceptable. La eficiencia de esta técnica consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para su futura utilización.

La Programación Dinámica no sólo tiene sentido aplicarla por razones de eficiencia, sino porque además presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado.

Donde tiene mayor aplicación la Programación Dinámica es en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

La solución de problemas mediante esta técnica se basa en el llamado principio de óptimo enunciado por Bellman en 1957 y que dice:

"*En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima*".

Hemos de observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión. Un ejemplo claro para el que no se verifica este principio aparece al tratar de encontrar el camino de coste máximo entre dos vértices de un grafo ponderado.

Para que un problema pueda ser abordado por esta técnica ha de cumplir dos condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de óptimo.

En grandes líneas, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

5.2 CÁLCULO DE LOS NÚMEROS DE FIBONACCI

Antes de abordar problemas más complejos veamos un primer ejemplo en el cual va a quedar reflejada toda esta problemática. Se trata del cálculo de los términos de la sucesión de números de Fibonacci. Dicha sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0,1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Por tanto, la forma más natural de calcular los términos de esa sucesión es mediante un programa recursivo:

```
PROCEDURE FibRec(n:CARDINAL):CARDINAL;
BEGIN
      IF n<=1 THEN RETURN 1
      ELSE
         RETURN FibRec(n-1) + FibRec(n-2)
      END
END FibRec;
```

El inconveniente es que el algoritmo resultante es poco eficiente ya que su tiempo de ejecución es de orden exponencial, como se vió en el primer capítulo.

Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y por tanto es necesario volver a calcular cada vez (véase el apartado 3.11 del capítulo 3, en donde se determina el número exacto de veces que se repite cada cálculo).

Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para poder reutilizarlos:

Fib(0)	*Fib*(1)	*Fib*(2)	...	*Fib*(n)

El algoritmo iterativo que calcula la sucesión de Fibonacci utilizando tal tabla es:

```
TYPE TABLA = ARRAY [0..n] OF CARDINAL
PROCEDURE FibIter(VAR T:TABLA;n:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      T[0]:=1;
      T[1]:=1:
      FOR i:=2 TO n DO
         T[i]:=T[i-1]+T[i-2]
      END;
      RETURN T[n]
   END
END FibIter;
```

Existe aún otra mejora a este algoritmo, que aparece al fijarnos que únicamente son necesarios los dos últimos valores calculados para determinar cada término, lo que permite eliminar la tabla entera y quedarnos solamente con dos variables para almacenar los dos últimos términos:

```
PROCEDURE FibIter2(n: CARDINAL):CARDINAL;
   VAR i,suma,x,y:CARDINAL; (* x e y son los 2 ultimos terminos *)
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      x:=1; y:=1;
      FOR i:=2 TO n DO
         suma:=x+y; y:=x; x:=suma;
      END;
      RETURN suma
   END
END FibIter2;
```

Aunque esta función sea de la misma complejidad temporal que la anterior (lineal), consigue una complejidad espacial menor, pues de ser de orden O(n) pasa a ser O(1) ya que hemos eliminado la tabla.

El uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos, pieza clave de los algoritmos de Programación Dinámica, hace que en

este capítulo nos fijemos no sólo en la complejidad temporal de los algoritmos estudiados, sino también en su complejidad espacial.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada, como veremos en alguno de los ejemplos de este capítulo.

5.3 CÁLCULO DE LOS COEFICIENTES BINOMIALES

En la resolución de un problema, una vez encontrada la expresión recursiva que define su solución, muchas veces la dificultad estriba en la creación del vector o la tabla que ha de conservar los resultados parciales. Así en este segundo ejemplo, aunque también sencillo, observamos que vamos a necesitar una tabla bidimensional algo más compleja. Se trata del cálculo de los coeficientes binomiales, definidos como:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ si } 0 < k < n, \quad \binom{n}{0} = \binom{n}{n} = 1.$$

El algoritmo recursivo que los calcula resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden O(nk) basado en la idea del Triángulo de Pascal. Para ello es necesario la creación de una tabla bidimensional en la que ir almacenando los valores intermedios que se utilizan posteriormente:

	0	1	2	3	...	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...	...	...	...	...	...		
...	...	...	...	...	...	...	
n-1						C(n-1,k-1) +	C(n-1,k)
n							C(n,k)

Iremos construyendo esta tabla por filas de arriba hacia abajo y de izquierda a derecha mediante el siguiente algoritmo de complejidad polinómica:

```
PROCEDURE CoefIter(n,k: CARDINAL):CARDINAL;
   VAR i,j: CARDINAL;
       C: TABLA;
BEGIN
```

```
    FOR i:=0 TO n DO C[i,0]:=1 END;
    FOR i:=1 TO n DO C[i,1]:=i END;
    FOR i:=2 TO k DO C[i,i]:=1 END;
    FOR i:=3 TO n DO
       FOR j:=2 TO i-1 DO
          IF j<=k THEN
             C[i,j]:=C[i-1,j-1]+C[i-1,j]
          END
       END
    END;
    RETURN C[n,k]
 END CoefIter.
```

5.4 LA SUBSECUENCIA COMÚN MÁXIMA

Hay muchos problemas para los cuales no sólo deseamos encontrar el valor de la solución óptima sino que además deseamos conocer cuál es la composición de esta solución, es decir, los elementos que forman parte de ella. En estos casos es necesario ir conservando no sólo los valores de las soluciones parciales, sino también cómo se llega a ellas. Esta información adicional puede ser almacenada en la misma tabla que las soluciones parciales, o bien en otra tabla al efecto.

Veamos un ejemplo en el que se crea una tabla y a partir de ella se reconstruye la solución. Se trata del cálculo de la subsecuencia común máxima. Vamos en primer lugar a definir el problema.

Dada una secuencia $X=\{x_1\ x_2\ ...\ x_m\}$, decimos que $Z=\{z_1\ z_2\ ...\ z_k\}$ es una subsecuencia de X (siendo $k \leq m$) si existe una secuencia creciente $\{i_1\ i_2\ ...\ i_k\}$ de índices de X tales que para todo j = 1, 2, ..., k tenemos $x_{ij} = z_j$.

Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común de X e Y si es subsecuencia de X y subsecuencia de Y. Deseamos determinar la subsecuencia de longitud máxima común a dos secuencias.

Solución (☺)

Llamaremos $L(i,j)$ a la longitud de la secuencia común máxima (SCM) de las secuencias X_i e Y_j, siendo X_i el i-ésimo prefijo de X (esto es, $X_i = \{x_1\ x_2\ ...\ x_i\}$) e Y_j el j-ésimo prefijo de Y, ($Y_j = \{y_1\ y_2\ ...\ y_j\}$).

Aplicando el principio de óptimo podemos plantear la solución como una sucesión de decisiones en las que en cada paso determinaremos si un carácter forma o no parte de la SCM. Escogiendo una estrategia hacia atrás, es decir, comenzando por los últimos caracteres de las dos secuencias X e Y, la solución viene dada por la siguiente relación en recurrencia:

$$L(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ L(i-1,j-1)+1 & \text{si } i\neq 0,\ j\neq 0 \text{ y } x_i = y_j \\ Max\{L(i,j-1), L(i-1,j)\} & \text{si } i\neq 0,\ j\neq 0 \text{ y } x_i \neq y_j \end{cases}$$

La solución recursiva resulta ser de orden exponencial, y por tanto Programación Dinámica va a construir una tabla con los valores $L(i, j)$ para evitar la repetición de cálculos. Para ilustrar la construcción de la tabla supondremos que X e Y son las secuencias de valores:

$$X = \{1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\}$$
$$Y = \{0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\}$$

La tabla que permite calcular la subsecuencia común máxima es:

		0	1	2	3	4	5	6	7	8
			1	0	0	1	0	1	0	1
0		0	0	0	0	0	0	0	0	0
1	0	0	0 Sup	1 Diag	1 Diag	1 Izq	1 Diag	1 Izq	1 Diag	1 Izq
2	1	0	1 Diag	1 Sup	1 Sup	2 Diag	2 Izq	2 Diag	2 Izq	2 Diag
3	0	0	1 Sup	2 Diag	2 Diag	2 Sup	3 Diag	3 Izq	3 Diag	3 Izq
4	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Izq	4 Diag
5	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag
6	0	0	1 Sup	2 Diag	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup
7	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
8	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
9	0	0	1 Sup	2 Diag	3 Diag	4 Sup	5 Diag	5 Sup	6 Diag	6 Sup

Esta tabla se va construyendo por filas y rellenando de izquierda a derecha. Como podemos ver en cada $L[i,j]$ hay dos datos: uno el que corresponde a la longitud de cada subsecuencia, y otro necesario para la construcción de la subsecuencia óptima.

La solución a la subsecuencia común máxima de las secuencias X e Y se encuentra en el extremo inferior derecho ($L[9,8]$) y por tanto su longitud es seis. Si queremos obtener cuál es esa subsecuencia hemos de recorrer la tabla (zona sombreada) a partir de esta posición siguiendo la información que nos indica cómo obtener las longitudes óptimas a partir de su procedencia (izquierda, diagonal o

superior). El algoritmo para construir la tabla tiene una complejidad de orden O(*nm*), siendo *n* y *m* las longitudes de las secuencias *X* e *Y*.

```
CONST N = ...; (* longitud maxima de una secuencia *)
TYPE  SECUENCIA = ARRAY [1..N] OF CARDINAL;
      PARES = RECORD numero:CARDINAL; procedencia:CHAR; END;
      TABLA = ARRAY [0..N],[0..N] OF PARES;

PROCEDURE SubSecMaxima(VAR X,Y:SECUENCIA;n,m:CARDINAL;VAR L:TABLA);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=0 TO m DO  (* condiciones iniciales *)
      L[i,0].numero:=0
   END;
   FOR j:=0 TO n DO
      L[0,j].numero:=0
   END;
   FOR i:=1 TO m DO
      FOR j:=1 TO n DO
         IF Y[i] = X[j] THEN
            L[i,j].numero:=L[i-1,j-1].numero+1;
            L[i,j].procedencia:="D"
         ELSIF L[i-1,j].numero >=L[i,j-1].numero THEN
            L[i,j].numero:=L[i-1,j].numero;
            L[i,j].procedencia:="S"
         ELSE
            L[i,j].numero:=L[i,j-1].numero;
            L[i,j].procedencia:="I"
         END
      END
   END
END SubSecMaxima.
```

Para encontrar cuál es esa subsecuencia óptima hacemos uso de la información contenida en el campo procedencia de la tabla *L*, sabiendo que '*I*'(por "Izq") significa que la información la toma de la casilla de la izquierda, '*S*' ("Sup") de la casilla superior y de la misma manera '*D*'("Diag") corresponde a la casilla que está en la diagonal. El algoritmo que recorre la tabla construyendo la solución a partir de esta información y de la secuencia *Y* es el que sigue:

```
PROCEDURE Escribir(VAR L:TABLA; VAR Y:SECUENCIA; i,j:CARDINAL;
                   VAR sol:SECUENCIA; VAR l:CARDINAL);
   (* sol es la secuencia solucion, l su longitud, i es la longitud
      de la secuencia Y, y j la de X *)
BEGIN
   IF (i=0) OR (j=0) THEN RETURN END;
```

```
    IF L[i,j].procedencia = "D" THEN
       Escribir(L,Y,i-1,j-1,sol,l);
       sol[l]:=Y[i];
       INC(l);
    ELSIF L[i,j].procedencia = "S" THEN
       Escribir(L,Y,i-1,j,sol,l)
    ELSE
       Escribir(L,Y,i,j-1,sol,l)
    END
 END Escribir
```

La complejidad de este algoritmo es de orden O(n+m) ya que en cada paso de la recursión puede ir disminuyendo o bien el parámetro i o bien j hasta alcanzar la posición $L[i,j]$ para $i = 0$ ó $j = 0$.

5.5 INTERESES BANCARIOS

Dadas n funciones f_1, f_2, ..., f_n y un entero positivo M, deseamos maximizar la función $f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$ sujeta a la restricción $x_1 + x_2 + ... + x_n = M$, donde $f_i(0) = 0$ (i=1,..,n), x_i son números naturales, y todas las funciones son monótonas crecientes, es decir, $x \geq y$ implica que $f_i(x) > f_i(y)$. Supóngase que los valores de cada función se almacenan en un vector.

Este problema tiene una aplicación real muy interesante, en donde f_i representa la función de interés que proporciona el banco i, y lo que deseamos es maximizar el interés total al invertir una cantidad determinada de dinero M. Los valores x_i van a representar la cantidad a invertir en cada uno de los n bancos.

Solución (☺)

Sea f_i un vector que almacena el interés del banco i ($1 \leq i \leq n$) para una inversión de 1, 2, 3, ..., M pesetas. Esto es, $f_i(j)$ indicará el interés que ofrece el banco i para j pesetas, con $0 < i \leq n$, $0 < j \leq M$.

Para poder plantear el problema como una sucesión de decisiones, llamaremos $I_n(M)$ al interés máximo al invertir M pesetas en n bancos,

$$I_n(M) = f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$$

que es la función a maximizar, sujeta a la restricción $x_1 + x_2 + ... + x_n = M$.

Veamos cómo aplicar el principio de óptimo. Si $I_n(M)$ es el resultado de una secuencia de decisiones y resulta ser óptima para el problema de invertir una cantidad M en n bancos, cualquiera de sus subsecuencias de decisiones ha de ser también óptima y así la cantidad

$$I_{n-1}(M - x_n) = f_1(x_1) + f_2(x_2) + ... + f_{n-1}(x_{n-1})$$

será también óptima para el subproblema de invertir ($M - x_n$) pesetas en $n - 1$ bancos. Y por tanto el principio de óptimo nos lleva a plantear la siguiente relación en recurrencia:

$$I_n(x) = \begin{cases} f_1(x) & \text{si } n = 1 \\ \underset{0 \le t \le x}{Max}\{I_{n-1}(x-t) + f_n(t)\} & \text{en otro caso.} \end{cases} \quad [5.1]$$

Para resolverla y calcular $I_n(M)$, vamos a utilizar una matriz I de dimensión nxM en donde iremos almacenando los resultados parciales y así eliminar la repetición de los cálculos. El valor de $I[i,j]$ va a representar el interés de j pesetas cuando se dispone de i bancos, por tanto la solución buscada se encontrará en $I[n,M]$. Para guardar los datos iniciales del problema vamos a utilizar otra matriz F, de la misma dimensión, y donde $F[i,j]$ representa el interés del banco i para j pesetas.

En consecuencia, para calcular el valor pedido de $I[n,M]$ rellenaremos la tabla por filas, empezando por los valores iniciales de la ecuación en recurrencia, y según el siguiente algoritmo:

```
CONST n = ...; (* numero de bancos *)
      M = ...; (* cantidad a invertir *)
TYPE MATRIZ = ARRAY [1..n],[0..M] OF CARDINAL;

PROCEDURE Intereses(VAR F:MATRIZ;VAR I:MATRIZ):CARDINAL;
   VAR i,j: CARDINAL;
BEGIN
   FOR i:=1 TO n DO I[i,0]:=0 END;
   FOR j:=1 TO M DO I[1,j]:=F[1,j] END;
   FOR i:=2 TO n DO
      FOR j:=1 TO M DO
         I[i,j]:=Max(I,F,i,j)
      END
   END;
   RETURN I[n,M]
END Intereses;
```

La función *Max* es la que calcula el máximo que aparece en la expresión [5.1]:

```
PROCEDURE Max(VAR I,F:MATRIZ;i,j:CARDINAL):CARDINAL;
   VAR max,t:CARDINAL;
BEGIN
   max:= I[i-1,j] + F[i,0];
   FOR t:=1 TO j DO
      max:=Max2(max,I[i-1,j-t]+F[i,t])
   END;
   RETURN max
END Max;
```

La función *Max2* es la que calcula el máximo de dos números naturales. La complejidad del algoritmo completo es de orden $O(nM^2)$, puesto que la complejidad de *Max* es $O(j)$ y se invoca dentro de dos bucles anidados que se

desarrollan desde 1 hasta M. Es importante hacer notar el uso de parámetros por referencia en lugar de por valor para evitar la copia de las matrices en la pila de ejecución del programa.

Por otro lado, la complejidad espacial del algoritmo es del orden $O(nM)$, pues de este orden son las dos matrices que se utilizan para almacenar los resultados intermedios.

En este ejemplo queda de manifiesto la efectividad del uso de estructuras en los algoritmos de Programación Dinámica para conseguir obtener tiempos de ejecución de orden polinómico, frente a los tiempos exponenciales de los algoritmos recursivos iniciales.

5.6 EL VIAJE MÁS BARATO POR RÍO

Sobre el río Guadalhorce hay n embarcaderos. En cada uno de ellos se puede alquilar un bote que permite ir a cualquier otro embarcadero río abajo (es imposible ir río arriba). Existe una tabla de tarifas que indica el coste del viaje del embarcadero i al j para cualquier embarcadero de partida i y cualquier embarcadero de llegada j más abajo en el río ($i < j$). Puede suceder que un viaje de i a j sea más caro que una sucesión de viajes más cortos, en cuyo caso se tomaría un primer bote hasta un embarcadero k y un segundo bote para continuar a partir de k. No hay coste adicional por cambiar de bote.

Nuestro problema consiste en diseñar un algoritmo eficiente que determine el coste mínimo para cada par de puntos i,j ($i < j$) y determinar, en función de n, el tiempo empleado por el algoritmo.

Solución (☺)

Llamaremos $T[i,j]$ a la tarifa para ir del embarcadero i al j (directo). Estos valores se almacenarán en una matriz triangular superior de orden n, siendo n el número de embarcaderos.

El problema puede resolverse mediante Programación Dinámica ya que para calcular el coste óptimo para ir del embarcadero i al j podemos hacerlo de forma recurrente, suponiendo que la primera parada la realizamos en un embarcadero intermedio k ($i < k \leq j$):

$$C(i,j) = T(i,k) + C(k,j).$$

En esta ecuación se contempla el viaje directo, que corresponde al caso en el que k coincide con j. Esta ecuación verifica también que la solución buscada $C(i,j)$ satisface el principio del óptimo, pues el coste $C(k,j)$, que forma parte de la solución, ha de ser, a su vez, óptimo. Podemos plantear entonces la siguiente expresión de la solución:

$$C(i,j) = \begin{cases} 0 & \text{si } i = j \\ \underset{i<k\leq j}{Min}\{T(i,k) + C(k,j)\} & \text{si } i < j \end{cases} \qquad [5.2]$$

La idea de esta segunda expresión surge al observar que en cualquiera de los trayectos siempre existe un primer salto inicial óptimo.

Para resolverla según la técnica de Programación Dinámica, hace falta utilizar una estructura para almacenar resultados intermedios y evitar la repetición de los cálculos. La estructura que usaremos es una matriz triangular de costes *C*[*i,j*], que iremos rellenando por diagonales mediante el procedimiento que hemos denominado *Costes*. La solución al problema es la propia tabla, y sus valores *C*[*i,j*] indican el coste óptimo para ir del embarcadero *i* al *j*.

```
CONST MAXEMBARCADEROS = ...;
TYPE MATRIZ=ARRAY[1..MAXEMBARCADEROS],[1..MAXEMBARCADEROS] OF CARDINAL;

   PROCEDURE Costes(VAR C:MATRIZ;n:CARDINAL);
      VAR i, diagonal:CARDINAL;
   BEGIN
      FOR i:=1 TO n DO C[i,i]:=0 END; (* condiciones iniciales *)
      FOR diagonal:=1 TO n-1 DO
         FOR i:=1 TO n-diagonal DO
            C[i,i+diagonal]:=Min(C,i,i+diagonal)
         END
      END
   END Costes;
```

Dicho procedimiento utiliza la siguiente función, que permite calcular la expresión del mínimo que aparece en la ecuación en recurrencia [5.2]:

```
   PROCEDURE Min(VAR C:MATRIZ; i,j:CARDINAL):CARDINAL;
      VAR k,min:CARDINAL;
   BEGIN
      min:=MAX(CARDINAL);
      FOR k:=i+1 TO j DO
         min:=Min2(min,T[i,k] + C[k,j])
      END;
      RETURN min
   END Min;
```

La función *Min2* es la que calcula el mínimo de dos números naturales. Es importante observar que esta función, por la forma en que se va rellenando la matriz *C*, sólo hace uso de los elementos calculados hasta el momento.

La complejidad del algoritmo es de orden $O(n^3)$, pues está compuesto por dos bucles anidados de tamaño *n*, que contienen la llamada a una función de orden $O(n)$, la que calcula el mínimo.

5.7 TRANSFORMACIÓN DE CADENAS

Sean *u* y *v* dos cadenas de caracteres. Se desea transformar *u* en *v* con el mínimo número de operaciones básicas del tipo siguiente: eliminar un carácter, añadir un

carácter, y cambiar un carácter. Por ejemplo, podemos pasar de *abbac* a *abcbc* en tres pasos:

abbac	→	*abac*	(eliminamos *b* en la posición 3)
	→	*ababc*	(añadimos *b* en la posición 4)
	→	*abcbc*	(cambiamos *a* en la posición 3 por *c*)

Sin embargo, esta tranformación no es óptima. Lo que queremos en este caso es diseñar un algoritmo que calcule el número mínimo de operaciones, de esos tres tipos, necesarias para transformar *u* en *v* y cuáles son esas operaciones, estudiando su complejidad en función de las longitudes de *u* y *v*.

Solución (☹)

En primer lugar, la transformación mostrada arriba no es óptima ya que podemos pasar de *abbac* a *abcbc* en sólo dos pasos:

abbac	→	*abcac*	(cambiamos *b* en la posicion 3 por *c*)
	→	*abcbc*	(cambiamos *a* en la posición 4 por *c*)

Llamaremos *m* a la longitud de la cadena *u*, *n* a la longitud de la cadena *v*, y *OB*(*m,n*) indicará el número de operaciones básicas mínimo para transformar una cadena *u* de longitud *m* en otra cadena *v* de longitud *n*.

Para resolver el problema utilizando Programación Dinámica es necesario plantearlo como una sucesión de decisiones que satisfaga el principio de óptimo.

Para plantearla, vamos a fijarnos en el último elemento de cada una de las cadenas. Si los dos son iguales, entonces tendremos que calcular el número de operaciones básicas necesarias para obtener de la primera cadena menos el último elemento, y la segunda cadena también sin el último elemento, es decir,

$$OB(m,n) = OB(m-1,n-1) \text{ si } u_m = v_n.$$

Pero si los últimos elementos fueran distintos habría que escoger la situación más beneficiosa de entre tres posibles: (*i*) considerar la primera cadena y la segunda pero sin el último elemento, o bien (*ii*) la primera cadena menos el último elemento y la segunda cadena, o bien (*iii*) las dos cadenas sin el último elemento. Esto da lugar a la siguiente relación en recurrencia para *OB*(*m*,*n*) para este caso:

$$OB(m,n)=1+Min\{OB(m,n-1),OB(m-1,\text{n}),OB(m-1,n-1)\} \text{ si } m\neq 0,\ n\neq 0 \text{ y } u_m\neq v_n.$$

En cuanto a las condiciones iniciales, tenemos las tres siguientes:

$$OB(0,0) = 0, \quad OB(m,0) = m \quad \text{y} \quad OB(0,n) = n.$$

Una vez disponemos de la ecuación en recurrencia necesitamos resolverla utilizando alguna estructura que nos permita reutilizar resultados intermedios, como mostramos a continuación:

```
CONST MAXCARACTERES = ...;
TYPE  CADENA=ARRAY[1..MAXCARACTERES] OF CHAR;
      TABLA=ARRAY[0..MAXCARACTERES],[0..MAXCARACTERES] OF CARDINAL;
```

```
PROCEDURE Cadena(VAR OB:TABLA;u,v:CADENA;n,m:CARDINAL):CARDINAL;
   VAR i,j: CARDINAL;
BEGIN
   FOR i:=0 TO m DO OB[i,0]:=i; END;
   FOR j:=0 TO n DO OB[0,j]:=j END;
   FOR i:=1 TO m DO
      FOR j:=1 TO n DO
         IF u[i]= v[j] THEN OB[i,j]:=OB[i-1,j-1]
         ELSE OB[i,j]:=Min3(OB[i,j-1],OB[i-1,j],OB[i-1,j-1])+1;
         END
      END
   END;
   RETURN OB[m,n]
END Cadena;
```

El procedimiento *Cadena* va a permitir la creación de la tabla *OB* que calcula el número mínimo de operaciones básicas. La solución se encuentra en *OB*[*m,n*], y la tabla se construye fila a fila (a partir de los valores que definen las condiciones iniciales) para poder ir reutilizando los valores calculados previamente. La función *Min3* es la que calcula el mínimo de tres enteros.

Como el algoritmo se limita a dos bucles anidados que sólo incluyen operaciones constantes la complejidad de este algoritmo es de orden O(*mn*).

5.8 LA FUNCIÓN DE ACKERMANN

La función de Ackermann se define recursivamente del modo siguiente:

$$\begin{cases} Ack\,(0,n) = n+1 \\ Ack\,(m,0) = Ack\,(m-1,1)\ si\ m>0 \\ Ack\,(m,n) = Ack\,(m-1, Ack\,(m,n-1))\ \ si\ \ m,n>0. \end{cases}$$

Nos planteamos los beneficios de diseñar, si es posible, un algoritmo de Programación Dinámica para calcular *Ack*(*m*,*n*).

Solución (☺)

Este problema nos permite analizar uno de los aspectos más importantes de la Programación Dinámica: la búsqueda de estructuras que permitan resolver una ecuación en recurrencia reutilizando los cálculos realizados hasta el momento. Como veremos en este ejemplo, esta tarea es a veces complicada.

Si observamos la definición recursiva de esta función para valores de *m* y *n* mayores que 0, vemos que para calcular el valor de *Ack*(*m*,*n*) será necesario utilizar un vector *A* suficientemente grande sobre el cual se irán almacenando de izquierda a derecha los sucesivos valores de la función, comenzando con *m* = 0. En cada paso

m del algoritmo se actualizarán los elementos $A[i]$ ($1 \le i \le n$), que van a almacenar el valor de $Ack(m,i)$.

Para el cálculo de cada elemento $A[i]$ se necesitará, además del elemento anterior (obsérvese que $A[i-1]$ contiene el valor de $Ack(m,i-1)$), un elemento del vector calculado en el paso anterior. La dificultad que entraña es que, conforme aumenta *m*, el elemento al que tenemos que referirnos del vector previamente calculado tiene un índice excesivamente elevado,

$$2^{2^{\cdot^{\cdot^{2^m}}}}$$

con lo cual la dimensión del vector *A* ha de ser muy grande.

Un algoritmo que resuelve el problema siguiendo estas indicaciones es el siguiente:

```
CONST MaxIndice = ...;
TYPE VECTOR = ARRAY[0..MaxIndice] OF CARDINAL;

PROCEDURE Ackerman(m,n:CARDINAL):CARDINAL;
   VAR i,j:CARDINAL; max,l:LONGREAL; A:VECTOR;
BEGIN
   max:=1.0;
   FOR i:=1 TO m DO
      max:=Pot(2.0,max)
   END;
   FOR i:=0 TO VAL(CARDINAL,Pot(max,LONGREAL(n))) DO
      A[i]:=i+1
   END;
   l:=max;
   FOR i:=1 TO m DO
      A[0]:=A[1];
      l:=Log(2.0,l);
      FOR j:=1 TO VAL(CARDINAL,Pot(l,LONGREAL(n))) DO
         A[j]:=A[A[j-1]]
      END;
   END;
   RETURN A[n]
END Ackerman;
```

La función *Pot(a,b)* es la que calcula la potencia *b*-ésima de un número *a* dado, esto es, a^b, y la función $Log(a,b)$ la que calcula el logaritmo en base *a* de *b*.

La complejidad temporal del algoritmo viene determinada en primer lugar por el valor del parámetro *m* ya que ha de actualizarse *m* veces el vector *A*, y además por el tamaño de este vector, resultando en un orden $O(m \cdot MaxIndice)$ debido a los dos bucles anidados del programa.

Este procedimiento es, sin embargo, muy "ingenuo". Y decimos esto porque, aunque perfectamente correcto desde un punto de vista teórico, su utilidad práctica es más bien poca. Al tener que manejar números tan grandes, que a su vez deben

ser usados como índices del vector, es muy pequeño el número de pasos que soporta sin exceder la capacidad de cálculo de cualquier ordenador. Desgraciadamente no conseguimos de esta forma manejar la "intratabilidad" de la función de Ackerman.

5.9 EL PROBLEMA DEL CAMBIO

Dentro del tema dedicado a algoritmos ávidos vimos un algoritmo para minimizar, dado un sistema monetario, el número de monedas necesarias para reunir una cantidad. Aquel algoritmo funcionaba cuando los tipos de monedas eran, por ejemplo, de 1, 5, 10 y 25 unidades, pero no obtenía necesariamente la descomposición óptima si añadíamos una moneda de 12 unidades al sistema.

Dado que el algoritmo ávido para este problema falla en algunas ocasiones, nos planteamos si puede resolverse utilizando Programación Dinámica de forma que la solución sea satisfactoria en todos los casos.

Solución (☹)

Sea n el número de tipos de monedas distintos, L la cantidad a conseguir y $T[1..n]$ un vector con el valor de cada tipo de moneda del sistema. Supondremos que disponemos de una cantidad inagotable de monedas de cada tipo.

Llamaremos $C(i,j)$ ($1 \le i \le n$, $1 \le j \le L$) al número mínimo de monedas para obtener la cantidad j restringiéndose a los tipos $T[1]$, $T[2]$, ..., $T[i]$. Si no se puede conseguir dicha cantidad entonces $C(i,j) = \infty$. En primer lugar hemos de encontrar una expresión recursiva de $C(i,j)$. Para ello observemos que en cada paso existen dos opciones:

1. No incluir ninguna moneda del tipo $T(i)$. Esto supone que el valor de $C(i,j)$ va a coincidir con el de $C(i-1,j)$, y por tanto $C(i,j) = C(i-1,j)$.
2. Sí incluirla. Pero entonces, al incluir la moneda del tipo $T(i)$, el número de monedas global coincide con el número óptimo de monedas para una cantidad $(j - T(i))$ más esta moneda $T(i)$ que se incluye, es decir podemos expresar $C(i,j)$ en este caso como $C(i,j) = 1 + C(i,j - T(i))$.

El cálculo de $C(i,j)$ óptimo tomará la solución más favorable, es decir, el menor valor de ambas opciones. Con esto, la relación en recurrencia queda definida como:

$$C(i,j) = \begin{cases} \infty & \text{si } i = 1 \text{ y } 1 \le j < T(i) \\ 0 & \text{si } j = 0 \\ 1 + C(i, j - T(i)) & \text{si } i = 1 \text{ y } j \ge T(i) \\ C(i-1, j) & \text{si } i > 1 \text{ y } j < T(i) \\ Min\{C(i-1, j), 1 + C(i, j - T(i))\} & \text{en otro caso} \end{cases}$$

Una vez disponemos de la solución recursiva del problema, aplicaremos un algoritmo de Programación Dinámica para calcular los $C(n,j)$, $1 \le j \le L$, mediante el uso de un vector de longitud L.

Llamemos *C* a dicho vector, que verifica que en cada paso *i* ($1 \le i \le n$), *C*[*j*] va a contener el valor de *C*(*i,j*). La idea es ir actualizando dicho vector paso a paso hasta llegar al paso *n*. El algoritmo que construye este vector es el siguiente:

```
CONST n = ...; (* num. tipos de monedas distintos del sistema *)
      L = ...; (* cantidad a conseguir *)
TYPE  TIPOMONEDA = ARRAY[1..n] OF CARDINAL;
      VECTOR = ARRAY[0..L] OF CARDINAL;

PROCEDURE Cambio(VAR C:VECTOR;L,n:CARDINAL;VAR T:TIPOMONEDA):CARDINAL;
   VAR i,j:CARDINAL;
BEGIN
   C[0]:=0;
   FOR i:=1 TO n DO
      FOR j:=1 TO L DO
         IF (i=1) AND (j<T[i]) THEN
            C[j]:=MAX(CARDINAL)
         ELSIF i=1 THEN
            C[j]:=1+C[j-T[1]]
         ELSIF j>=T[i] THEN
            C[j]:=Min2(C[j],1+C[j-T[i]])
         (* ELSE C[j] no se modifica *)
         END
      END;
   END;
   RETURN C[L]
END Cambio;
```

El algoritmo devuelve un valor, que es el número óptimo de monedas necesario para obtener la cantidad *L*, siendo su complejidad temporal de orden O(*nL*) y la espacial de orden O(*L*).

Una vez calculado el vector que nos permite encontrar el número mínimo de monedas es posible diseñar un algoritmo que construya la solución.

Para ello va a ser necesario mantener una tabla de valores lógicos *P*[*i,j*] que indique la procedencia del valor *C*(*i,j*) en la expresión en recurrencia, es decir, si $C(i,j) = C(i-1,j)$ (lo que indica que no se toma una moneda de valor *T*[*i*]) o bien $C(i,j) = C(i,j-T[i]) + 1$ (indicando que sí se incluye una moneda del valor *T*[*i*]). Por tanto, bastará definir *P*[*i,j*] = *FALSE* en el primer caso, *TRUE* en el segundo.

Para poder calcular los valores de esta matriz de procedencia *P*[*i,j*] se necesitan tener presentes todos los valores de *C*(*i,j*), para lo cual ya no es suficiente un vector *C* como el utilizado en el apartado anterior, sino que será necesaria la creación de una matriz *C*[*i,j*] que conserve los distintos valores para $i = 1,2,...,n$. El algoritmo que implementa tal estrategia es el siguiente:

```
TYPE MONEDAS = ARRAY[1..n] OF CARDINAL;
     MATRIZ = ARRAY[1..n],[0..L] OF BOOLEAN;
     CAMBIO = ARRAY[1..n],[0..L] OF CARDINAL;
```

```
PROCEDURE Procedencia(VAR P:MATRIZ; VAR C:CAMBIO; L,n:CARDINAL;
                      T:MONEDAS);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      P[i,0]:=FALSE;
      C[i,0]:=0
   END;
   FOR i:=1 TO n DO
      FOR j:=1 TO L DO
         IF (i=1) AND (j<T[i]) THEN
            C[i,j]:=MAX(CARDINAL);
            P[i,j]:=FALSE
         ELSIF i=1 THEN
            C[i,j]:=1 + C[i,j-T[1]];
            P[i,j]:=TRUE
         ELSIF j<T[i] THEN
            C[i,j]:=C[i-1,j];
            P[i,j]:=FALSE
         ELSE
            C[i,j]:=Min2(C[i-1,j],1+C[i,j-T[i]]);
            P[i,j]:=(C[i,j] <> C[i-1,j])
         END
      END
   END
END Procedencia;
```

La solución se encuentra recorriendo la tabla *P* en sentido inverso, comenzando por el valor *P*[*n,L*], como se muestra en el siguiente algoritmo:

```
TYPE NUMMONEDAS = ARRAY[1..n] OF CARDINAL;

PROCEDURE Monedas(P:MATRIZ;C:CAMBIO;L,n:CARDINAL;T:MONEDAS):NUMMONEDAS;
   VAR monedas:NUMMONEDAS; ind,i,j:CARDINAL;
BEGIN
   i:=n; j:=L;
   FOR ind:=1 TO n DO monedas[ind]:=0 END;
   WHILE (i<>0) AND (j<>0) DO
      IF P[i,j]=FALSE THEN DEC(i)
      ELSE monedas[i]:=monedas[i]+1; j:=j-T[i]
      END
   END;
   IF i=0 THEN monedas[1]:=C[i,j]+monedas[1] END;
   RETURN monedas
END Monedas;
```

La resolución del problema requiere por una parte el algoritmo *Procedencia* para la construcción de la tabla P que permite conocer el número de monedas, y por tanto cuando se trata de n tipos de monedas diferentes y una cantidad L, su orden de complejidad es $O(nL)$.

Además, para conocer la solución es necesario recorrer la tabla desde la posición $P[n,L]$ hasta $P[0,0]$ a través de $n-1$ pasos de orden de complejidad $O(n)$ para llegar a la fila 1. Asimismo hay que tener en cuenta los pasos que hay que dar de derecha a izquierda hasta llegar a la columna cero, que viene dado por el número de monedas que intervienen en la solución, es decir, por el valor de $C[n,L]$. Podemos concluir por tanto que su complejidad es $O(n+C[n,L])$.

5.10 EL ALGORITMO DE DIJKSTRA

Sea un grafo ponderado $g = (V,A)$, donde V es su conjunto de vértices, A el conjunto de arcos y sea $L[i,j]$ su matriz de adyacencia. Queremos calcular el camino más corto entre un vértice v_i tomado como origen y cada vértice restante v_j del grafo.

El clásico algoritmo de Dijkstra trabaja en etapas, en donde en cada una de ellas va añadiendo un vértice al conjunto D que representa aquellos vértices para los que se conoce su distancia al vértice origen. Inicialmente el conjunto D contiene sólo al vértice origen.

Aún siendo el algoritmo de Dijkstra un claro ejemplo de algoritmo ávido, nos preguntamos si puede ser planteado como un algoritmo de Programación Dinámica, y si de ello se deriva alguna ventaja.

Solución (☺)

La técnica de la Programación Dinámica tiene grandes ventajas, y una de ellas es la de ofrecer un diseño adecuado y eficiente a todos los problemas que puedan plantearse de forma recursiva y cumplan el principio del óptimo.

Así, es posible plantear el algoritmo de Dijkstra en términos de la Programación Dinámica, y de esta forma aprovechar el método de diseño y las ventajas que esta técnica ofrece.

En primer lugar, observemos que es posible aplicar el principio de óptimo en este caso: si en el camino mínimo de v_i a v_j está un vértice v_k como intermedio, los caminos parciales de v_i a v_k y de v_k a v_j han de ser a su vez mínimos.

Llamaremos $D(j)$ al vector que contiene el camino mínimo desde el vértice origen $i = 1$ a cada vértice v_j, $2 \le j \le n$, siendo n el número de vértices. Inicialmente D contiene los arcos $L(1,j)$, o bien ∞ si no existe el arco. A continuación, y para cada vértice v_k del grafo con $k \neq 1$, se repetirá:

$$D(j) = \underset{1<k\le n}{Min}\{D(j), D(k) + L(k,j)\} \qquad [5.3]$$

De esta forma el algoritmo que resuelve el problema puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
     MARCA = ARRAY [1..n] OF BOOLEAN;(* elementos ya considerados*)
     SOLUCION = ARRAY [2..n] OF CARDINAL;

PROCEDURE Dijkstra(VAR L:MATRIZ;VAR D:SOLUCION);
   VAR i,j,menor,pos,s:CARDINAL; S:MARCA;
BEGIN
   FOR i:=2 TO n DO
      S[i]:=FALSE;
      D[i]:=L[1,i]
   END;
   S[1]:=TRUE;
   FOR i:=2 TO n-1 DO
      menor:=Menor(D,S,pos);
      S[pos]:=TRUE;
      FOR j:=2 TO n DO
         IF NOT(S[j]) THEN
            D[j]:= Min2(D[j],D[pos]+L[pos,j])
         END;
      END;
   END
END Dijkstra;
```

La función *Menor* es la que calcula el mínimo de la expresión en recurrencia [5.3] que define la solución del problema:

```
PROCEDURE Menor(VAR D:SOLUCION; VAR S:MARCA; VAR pos:CARDINAL)
                                                       :CARDINAL;
   VAR menor,i:CARDINAL;
BEGIN
   menor:=MAX(CARDINAL); pos:=1;
   FOR i:=2 TO n DO
      IF NOT(S[i]) THEN
         IF D[i]<menor THEN
            menor:=D[i]; pos:=i
         END
      END
   END;
   RETURN menor
END Menor;
```

La complejidad temporal del algoritmo es de orden O(n^2), siendo de orden O(n) su complejidad espacial. No ganamos sustancialmente en eficiencia mediante el uso de esta técnica frente al planteamiento ávido del algoritmo, pero sin embargo sí

ganamos en sencillez del diseño e implementación de la solución a partir del planteamiento del problema.

5.11 EL ALGORITMO DE FLOYD

Sea g un grafo dirigido y ponderado. Para calcular el menor de los caminos mínimos entre dos vértices cualesquiera del grafo, podemos aplicar el algoritmo de Dijkstra a todos los pares posibles y calcular su mínimo, o bien aplicamos el siguiente algoritmo (Floyd) que, dada la matriz L de adyacencia del grafo g, calcula una matriz D con la longitud del camino mínimo que une cada par de vértices:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE  MATRIZ = ARRAY[1..n],[1..n] OF CARDINAL;

PROCEDURE Floyd (VAR L,D:MATRIZ);
   VAR i,j,k: CARDINAL;
BEGIN
   FOR i:=1 TO n DO FOR j:=1 TO n DO
         D[i,j]:=L[i,j]
   END END;
   FOR k:=1 TO n DO
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            D[i,j]:=Min2(D[i,j],D[i,k]+D[k,j])
         END
      END
   END
END Floyd;
```

Nos planteamos si tal algoritmo puede ser considerado o no de Programación Dinámica, es decir, si reune las características esenciales de ese tipo de algoritmos.

Solución (☺)

Este algoritmo puede ser considerado de Programación Dinámica ya que es aplicable el principio de óptimo, que puede enunciarse para este problema de la siguiente forma: si en el camino mínimo de v_i a v_j, v_k es un vértice intermedio, los caminos de v_i a v_k y de v_k a v_j han de ser a su vez caminos mínimos. Por lo tanto, puede plantearse la relación en recurrencia que resuelve el problema como:

$$D_k(i,j) = \underset{k \geq 1}{Min}\{D_{k-1}(i,k), D_{k-1}(k,j)\}$$

Tal ecuación queda resuelta mediante el algoritmo presentado que, siguiendo el esquema de la Programación Dinámica, utiliza una matriz para evitar la repetición de los cálculos. Con ello consigue que su complejidad temporal sea de orden $O(n^3)$ debido al triple bucle anidado en cuyo interior hay tan sólo operaciones constantes.

5.12 EL ALGORITMO DE WARSHALL

Al igual que ocurre con el algoritmo de Floyd descrito en el apartado anterior, estamos interesados en encontrar caminos entre cada dos vértices de un grafo. Sin embargo, aquí no nos importa su longitud, sino sólo su existencia. Por tanto, lo que deseamos es diseñar un algoritmo que permita conocer si dos vértices de un grafo están conectados o no, lo que nos llevaría al cierre transitivo del grafo.

Solución (☺)

Para un grafo $g = (V,A)$ cuya matriz de adyacencia sea L, el algoritmo pedido puede ser implementado como sigue:

```
CONST n = ...; (* numero de vertices del grafo *)
TYPE  MATRIZ = ARRAY[1..n],[1..n] OF BOOLEAN;

PROCEDURE Warshall (VAR L,D:MATRIZ);
   VAR i,j,k: CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      FOR j:=1 TO n DO
         D[i,j]:=L[i,j]
      END
   END;
   FOR k:=1 TO n DO
      FOR i:=1 TO n DO
         FOR j:=1 TO n DO
            D[i,j]:=D[i,j] OR (D[i,k] AND D[k,j])
         END
      END
   END
END Warshall;
```

Tras la ejecución del algoritmo, la solución se encuentra en la matriz D, que verifica que $D[i,j] = TRUE$ si y sólo si existe un camino entre los vértices i y j. Obsérvese la similitud entre este algoritmo y el de Floyd. En cuanto a su complejidad, podemos afirmar que es de orden $O(n^3)$ debido al triple bucle anidado que posee, en cuyo interior sólo se realizan operaciones constantes.

5.13 ORDENACIONES DE OBJETOS ENTRE DOS RELACIONES

Dados n objetos, queremos calcular el número de ordenaciones posibles según las relaciones “<” e “=”. Por ejemplo, dados tres objetos A, B y C, el algoritmo debe determinar que existen 13 ordenaciones distintas: A=B=C, A=B<C, A<B=C, A<B<C, A<C<B, A=C<B, B<A=C, B<A<C, B<C<A, B=C<A, C<A=B, C<A<B y C<B<A.

Solución (☞)

Llamaremos C_n al número de ordenaciones posible con n objetos. Si $1 \le k \le n$, podemos expresar C_k como:

$$C_k = I_0^{(k)} + I_1^{(k)} + ... + I_{k-1}^{(k)} = \sum_{j=0}^{k-1} I_j^{(k)}$$

siendo $I_j^{(k)}$ el número de formas posibles de poner k elementos en donde hay j símbolos "=" (es decir, $k - j - 1$ símbolos distintos), con $0 \le j < k$, $1 \le k \le n$. Vamos a tratar de expresar C_k en función de C_{k-1}. Para ello, supongamos que ya tenemos

$$C_{k-1} = \sum_{j=0}^{k-2} I_j^{(k-1)}$$

y añadimos un nuevo elemento. Pueden ocurrir dos casos: que sea distinto a todos los k–1 elementos anteriores, o bien que sea igual a uno de ellos. Entonces, la expresión de C_k va a venir dada por:

$$C_k = (kI_0^{(k-1)} + (k-1)I_1^{(k-1)} + (k-2)I_2^{(k-1)} + ... + 2I_{k-2}^{(k-1)}) +$$
$$((k-1)I_0^{(k-1)} + (k-2)I_1^{(k-1)} + ... + 2I_{k-3}^{(k-1)} + I_{k-2}^{(k-1)}) = \sum_{j=0}^{k-2}(2(k-j)-1)I_j^{(k-1)}$$

Con esto, tenemos C_k en función de los $I_j^{(k-1)}$, es decir, de los componentes del caso anterior. Ahora bien, es posible también relacionar los $I^{(k)}$ con los $I^{(k-1)}$ de la siguiente manera:

$$\begin{aligned}
I_0^{(k)} &= kI_0^{(k-1)} \\
I_1^{(k)} &= (k-1)I_1^{(k-1)} + (k-1)I_0^{(k-1)} \\
I_2^{(k)} &= (k-2)I_2^{(k-1)} + (k-2)I_1^{(k-1)} \\
&... \\
I_{k-2}^{(k)} &= 2I_{k-2}^{(k-1)} + 2I_{k-3}^{(k-1)} \\
I_{k-1}^{(k)} &= I_{k-2}^{(k-1)}
\end{aligned}$$

cuyas condiciones iniciales son $I_0^{(2)} = 2, I_1^{(2)} = 1$. Esto también puede expresarse como sigue:

$$I_j^{(k)} = (k-j)(I_j^{(k-1)} + I_{j-1}^{(k-1)}) \quad \text{para } 0 \le j \le k-1 \text{ y } 2 \le k \le n$$
$$I_0^{(2)} = 2$$
$$I_1^{(2)} = 1$$
$$I_{-1}^{(k)} = 0$$
$$I_k^{(k)} = 0$$

Así, el problema puede resolverse calculando cada $I_j^{(n)}$ ($0 \le j \le n-1$), para finalmente calcular C_n mediante la expresión:

$$C_n = \sum_{j=0}^{n-1} I_j^{(n)}$$

El algoritmo que implementa tal estrategia es el siguiente:

```
CONST n = ...; (* numero de objetos *)
TYPE VECTOR = ARRAY [-1..n] OF INTEGER;

PROCEDURE Ordenaciones(VAR I:VECTOR; n:INTEGER):INTEGER;
   VAR x,y,s,k,j:INTEGER;
BEGIN
   IF n <= 1 THEN RETURN n END; (* caso base *)
   FOR j:=-1 TO n DO I[j]:=0 END; (* inicializamos el vector I *)
   I[0]:=1; x:=0;
   FOR k:=2 TO n DO
      FOR j:=0 TO n-1 DO
         IF j>1 THEN I[j-2]:=y END;
         y:=x;
         x:=(k-j)*(I[j]+I[j-1])
      END;
      I[n-2]:=y; I[n-1]:=x;
   END;
   s:=0;
   FOR j:=0 TO n-1 DO
      s:=s+I[j]
   END;
   RETURN s
END Ordenaciones;
```

Respecto a su complejidad espacial, tan sólo utiliza el vector *I* por lo que es de orden O(n). Y en cuanto a su complejidad temporal, el algoritmo utiliza dos bucles anidados para el cálculo de los valores del vector, por lo que podemos afirmar que su orden es O(n^2).

5.14 EL VIAJANTE DE COMERCIO

¿Podría aplicarse la técnica de Programación Dinámica al problema del viajante de comercio? Recordemos que este problema consistía en encontrar el camino sin ciclos de menor peso de un grafo ponderado que recorra todos los vértices y vuelva al vértice original.

Solución (☺)

En primer lugar, vamos a plantear la solución del problema como una sucesión de decisiones que verifique el principio de óptimo. La idea va a consistir en construir una solución mediante la búsqueda sucesiva de recorridos mínimos de tamaño 1, 2, 3, etc.

Representando el problema a través de un grafo $g = (V,A)$, y siendo L su matriz de adyacencia, cada recorrido del viajante que parte del vértice v_1 estará formado por un arco (v_1,v_k) para algún vértice v_k perteneciente a $V–\{v_1\}$ y un camino de v_k al vértice v_1.

Pero si el recorrido es óptimo también ha de ser óptimo el camino de v_k al vértice v_1, pues si no lo fuese llegaríamos a una contradicción. Si no lo fuese y existiera otro camino mejor, incluyendo a éste en el recorrido original obtendríamos un camino mejor que el óptimo, lo cual es imposible. Por tanto, se cumple el principio de óptimo.

Planteemos entonces la relación en recurrencia. Para ello, llamaremos $D(v_i,S)$ a la longitud del camino mínimo que partiendo del vértice v_i pasa por todos los vértices del conjunto S y vuelve al vértice v_i. La solución al problema del viajante vendrá dada entonces por $D(v_1,V–\{v_1\})$:

$$D(v_i, V-\{v_1\}) = \underset{2\le k\le n}{Min}\{L(v_1,v_k) + D(v_k, V-\{v_1,v_k\}\}$$

Generalizando para comenzar el recorrido desde cualquier vértice:

$$D(v_i, V) = \underset{i\notin V,\, j\in V}{Min}\{L(v_i,v_j) + D(v_j, V-v_j)\}$$

$$D(v_i, \{\}) = L(v_i, v_1) \text{ para } 1 \le i \le n$$

Obsérvese la diferencia que existe entre la estrategia de este algoritmo y los que tratamos de diseñar siguiendo dos técnicas ávidas (ver el problema 4.4 del capítulo anterior). En los algoritmos ávidos se ha de escoger una de las posibles opciones en cada paso, y una vez tomada –o descartada–, ya no vuelve a ser considerada nunca. Son algoritmos que no guardan "historia", y por tanto no siempre funcionan. Sin embargo, en la Programación Dinámica la solución al problema total se va construyendo de otra forma: a partir de las soluciones óptimas para problemas más pequeños.

No obstante, el diseño aquí realizado tiene un serio inconveniente: su implementación utilizando una estructura de datos que permita reutilizar los cálculos. Tal estructura debería contener las soluciones intermedias necesarias para el cómputo de $D(v_1,V–\{v_1\})$, pero estas son demasiadas.

En efecto, la tabla debe tener n filas, y 2^n columnas, pues éste es el cardinal de las partes del conjunto V, que son todas las posibilidades que puede tomar el segundo parámetro de D en su definición.

Por tanto, sí existe una solución al problema del viajante utilizando Programación Dinámica, pero no sólo no consigue mejorar la eficiencia de su versión clásica mediante Vuelta Atrás (véase el siguiente capítulo), sino que tampoco ofrece una mejora en cuanto a la simplicidad de su implementación.

5.15 HORARIOS DE TRENES

Una compañía de ferrocarriles sirve n estaciones $S_1,...,S_n$ y trata de mejorar su servicio al cliente mediante terminales de información. Dadas una estación origen S_o y una estación destino S_d, un terminal debe ofrecer (inmediatamente) la información sobre el horario de los trenes que hacen la conexión entre S_o y S_d y que minimizan el tiempo de trayecto total.

Necesitamos implementar un algoritmo que realice esta tarea a partir de la tabla con los horarios, suponiendo que las horas de salida de los trenes coinciden con las de sus llegadas (es decir, que no hay tiempos de espera) y que, naturalmente, no todas las estaciones están conectadas entre sí por líneas directas; así, en muchos casos hay que hacer transbordos aunque se supone que tardan tiempo cero en efectuarse.

Solución (☺)

Llamaremos $T(i,j,V)$ al tiempo del trayecto mínimo para ir de la estación de origen i a la estación destino j, pudiendo utilizar como estaciones intermedias las contenidas en el conjunto V, y llamaremos $L(i,j)$ al tiempo del trayecto directo de i a j, siendo ∞ si esta conexión no existe. De forma análoga a como razonábamos para el problema de los embarcaderos sobre el río (apartado 5.6), podemos plantear la solución al problema mediante una ecuación en recurrencia:

$$T(i,j,V) = \underset{k \in V, k \neq i, k \neq j}{Min} \{L(i,j), T(k,j,V-k) + L(i,k)\}$$

Esta ecuación trata de comprobar si es más beneficioso ir de forma directa o a través de cada uno de los posibles caminos. Esto hay que hacerlo para cada par (i,j) desde 1 hasta n. Obsérvese cómo llegamos a ella por el principio de óptimo pues cualquier subtrayecto de un trayecto óptimo a de ser, a su vez, óptimo.

Podemos representar nuestro problema mediante un grafo siendo las estaciones los vértices del grafo y las aristas las conexiones entre dos estaciones, pudiendo no existir si no hay trayecto directo entre ambas. La solución al problema se puede alcanzar resolviendo el algoritmo de Dijkstra para cada vértice del grafo, puesto que tal algoritmo calcula los caminos mínimos desde un único origen hasta los demás vértices en un grafo.

Respecto a su complejidad, ésta coincide con la del algoritmo de Dijkstra, que es aceptable para un número n de estaciones razonablemente grande.

5.16 LA MOCHILA (0,1)

En el apartado 4.7 del capítulo anterior se planteó el problema de la Mochila (0,1), que consistía en decidir de entre n objetos de pesos p_1, p_2,..., p_n y beneficios b_1, b_2,..., b_n, cuáles hay que incluir en una mochila de capacidad M sin superar dicha capacidad y de forma que se maximice la suma de los beneficios de los elementos escogidos. Los algoritmos ávidos planteados entonces no conseguían resolver el problema. Nos cuestionamos aquí si este problema admite una solución mediante Programación Dinámica.

Solución (☹)

Para encontrar un algoritmo de Programación Dinámica que lo resuelva, primero hemos de plantear el problema como una secuencia de decisiones que verifique el principio de óptimo. De aquí seremos capaces de deducir una expresión recursiva de la solución. Por último habrá que encontrar una estructura de datos adecuada que permita la reutilización de los cálculos de la ecuación en recurrencia, consiguiendo una complejidad mejor que la del algoritmo puramente recursivo.

Siendo M la capacidad de la mochila y disponiendo de n elementos, llamaremos $V(i,p)$ al valor máximo de la mochila con capacidad p cuando consideramos i objetos, con $0 \leq p \leq M$ y $1 \leq i \leq n$. La solución viene dada por el valor de $V(n,M)$. Denominaremos d_1, d_2, ..., d_n a la secuencia de decisiones que conducen a obtener $V(n,M)$, donde cada d_i podrá tomar uno de los valores 1 ó 0, dependiendo si se introduce o no el i-ésimo elemento. Podemos tener por tanto dos situaciones distintas:

- Que $d_n = 1$. La subsecuencia de decisiones d_1, d_2, ..., d_{n-1} ha de ser también óptima para el problema $V(n-1,M-p_n)$, ya que si no lo fuera y existiera otra subsecuencia e_1, e_2, ..., e_{n-1} óptima, la secuencia e_1, e_2, ..., e_{n-1}, d_n también sería óptima para el problema $V(n,M)$ lo que contradice la hipótesis.
- Que $d_n = 0$. Entonces la subsecuencia decisiones d_1, d_2, ..., d_{n-1} ha de ser también óptima para el problema $V(n-1,M)$.

Podemos aplicar por tanto el principio de óptimo para formular la relación en recurrencia. Teniendo en cuenta que en la mochila no puede introducirse una fracción del elemento sino que el elemento i se introduce o no se introduce, en una situación cualquiera $V(i,p)$ tomará el valor mayor entre $V(i-1,p)$, que indica que el elemento i no se introduce, y $V(i-1,p-p_i)+b_i$, que es el resultado de introducirlo y de ahí que la capacidad ha de disminuir en p_i y el valor aumentar en b_i, y por tanto podemos plantear la solución al problema mediante la siguiente ecuación:

$$V(i,p) = \begin{cases} 0 & \text{si } i = 0 \text{ y } p \geq 0 \\ -\infty & \text{si } p < 0 \\ Max\{V(i-1,p), V(i-1,p-p_i)+b_i\} & \text{en otro caso.} \end{cases}$$

Estos valores se van almacenando en una tabla construida mediante el algoritmo:

```
TYPE TABLA = ARRAY[1..n],[0..M] OF CARDINAL;
      DATOS = RECORD peso,valor:CARDINAL; END;
      TIPOOBJETO = ARRAY[1..n] OF DATOS;

PROCEDURE Mochila (i,p:CARDINAL; VAR obj:TIPOOBJETO):CARDINAL;
   VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
   FOR elem:=1 TO i DO
      V[elem,0]:=0;
      FOR cap:=1 TO p DO
         IF (elem=1) AND (cap<obj[1].peso) THEN
            V[elem,cap]:=0
         ELSIF elem=1 THEN
            V[elem,cap]:=obj[1].valor
         ELSIF cap<obj[elem].peso THEN
            V[elem,cap]:=V[elem-1,cap]
         ELSE V[elem,cap]:=
    Max2(V[elem-1,cap],obj[elem].valor+V[elem-1,cap-obj[elem].peso])
         END
      END
   END;
   RETURN V[i,p]
END Mochila;
```

El problema se resuelve invocando a la función con $i=n$, $p=M$. La complejidad del algoritmo viene determinada por la construcción de una tabla de dimensiones $n\times M$ y por tanto su tiempo de ejecución es de orden de complejidad $O(nM)$. La función *Max*2 es la que calcula el máximo de dos valores.

Si además del valor de la solución óptima se desea conocer los elementos que son introducidos, es decir, la composición de la mochila, es necesario añadir al algoritmo la construcción de una tabla de valores lógicos que indique para cada valor $E[i,j]$ si el elemento i forma parte de la solución para la capacidad j o no:

```
TYPE ENTRAONO = ARRAY[1..n],[0..P] OF BOOLEAN;

PROCEDURE Max2especial(x,y:CARDINAL;VAR esmenorx:BOOLEAN):CARDINAL;
BEGIN
   IF x>y THEN
      esmenorx:=FALSE;
      RETURN x
   ELSE
      esmenorx:=TRUE;
      RETURN y
   END
END Max2especial;
```

```
PROCEDURE Mochila2(i,p:CARDINAL;obj:TIPOOBJETO;VAR E:ENTRAONO)
                                                    :CARDINAL;
   VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
   FOR elem:=1 TO i DO
      V[elem,0]:=0;
      FOR cap:=1 TO p DO
         IF (elem=1) AND (cap<obj[1].peso) THEN
               V[elem,cap]:=0;
               E[elem,cap]:=FALSE
         ELSIF elem=1 THEN
               V[elem,cap]:=obj[1].valor;
               E[elem,cap]:=TRUE
         ELSIF cap<obj[elem].peso THEN
               V[elem,cap]:=V[elem-1,cap];
               E[elem,cap]:=FALSE
         ELSE  V[elem,cap]:=Max2especial(V[elem-1,cap],
          obj[elem].valor+V[elem-1,elem-obj[elem].peso],E[elem,cap]);
         END
      END
   END;
   RETURN V[i,p]
END Mochila2;
```

Por otra parte, es necesario construir un algoritmo que interprete los valores de esta tabla para componer la solución. Esto se realizará recorriéndola en sentido inverso desde los valores $i = n$, $j = M$ hasta $i = 0$, $j = 0$, mediante el siguiente algoritmo:

```
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;

PROCEDURE Componer(VAR sol:SOLUCION);
   VAR elem,cap:CARDINAL;
BEGIN
   FOR elem:=1 TO n DO (* inicializa solucion *)
      sol[elem]:=0
   END;
   elem:= n; cap:= M;
   WHILE (elem<>0) AND (cap<>0) DO
      IF entra[elem,cap] THEN
         sol[elem]:=1;
         cap:=cap-obj[elem].peso
      END;
      DEC(elem)
   END
 END Componer;
```

5.17 LA MOCHILA (0,1) CON MÚLTIPLES ELEMENTOS

Este problema se basa en el de la Mochila (0,1) pero en vez de existir *n* objetos distintos, de lo que disponemos es de *n* tipos de objetos distintos. Con esto, de un objeto cualquiera podemos escoger tantas unidades como deseemos.

Este problema se puede formular también como una modificación al problema de la Mochila (0,1), en donde sustituimos el requerimiento de que $x_i = 0$ ó $x_i = 1$, por el que x_i sean números naturales. Como en el problema original, deseamos maximizar la suma de los beneficios de los elementos introducidos, sujeta a la restricción de que éstos no superen la capacidad de la mochila.

Solución (☺)

Para encontrar un algoritmo de Programación Dinámica que resuelva el problema, primero hemos de plantearlo como una secuencia de decisiones que verifiquen el principio del óptimo. De aquí seremos capaces de deducir una expresión recursiva de la solución. Por último habrá que encontrar una estructura de datos adecuada que permita la reutilización de los cálculos de la ecuación en recurrencia, consiguiendo una complejidad mejor que la del algoritmo puramente recursivo.

Con esto en mente, llamaremos $V(i,p)$ al valor máximo de una mochila de capacidad *p* y con *i* tipos de objetos. Iremos decidiendo en cada paso si introducimos o no un objeto de tipo *i*. Por consiguiente, para calcular $V(i,p)$ existen dos opciones en cada paso:

- No introducir ninguna unidad del tipo *i*, con lo cual el valor de la mochila $V(i,p)$ es el calculado para $V(i–1,p)$.
- Introducir una unidad más del objeto *i* lo cual indica que el valor de $V(i,p)$ será el resultado obtenido para $V(i,p–p_i)$ más el valor del objeto v_i, con lo cual se verifica que $V(i,p) = V(i,p–p_i) + b_i$.

Esto nos permite establecer la siguiente relación en recurrencia para $V(i,p)$:

$$V(i,p) = \begin{cases} 0 & \text{si} \quad p = 0 \\ (p \div p_i)b_i & \text{si} \quad i = 1 \\ V(i-1,p) & \text{si} \quad p < p_i \\ Max\{V(i-1,p), V(i,p-p_i)+b_i\} & \text{en otro caso.} \end{cases}$$

Utilizaremos una matriz *n*x*M* para almacenar los valores de *V* que vayamos obteniendo y así no repetir cálculos. El algoritmo que resuelve el problema es el siguiente:

```
TYPE TABLA = ARRAY[1..n],[0..M] OF CARDINAL;
     DATOS = RECORD peso,valor:CARDINAL; END;
     TIPOOBJETO = ARRAY[1..n] OF DATOS;
```

```
PROCEDURE Mochila3(i,p:CARDINAL;VAR obj:TIPOOBJETO):CARDINAL;
   VAR elem,cap:CARDINAL; V:TABLA;
BEGIN
   FOR elem:=1 TO i DO (* condiciones iniciales *)
      V[elem,0]:=0
   END;
   FOR cap:=1 TO p DO
      V[1,cap]:=(cap DIV obj[1].peso)*obj[1].valor
   END;
   FOR elem:=2 TO num DO
      FOR cap:=1 TO p DO
         IF (cap < p[elem]) THEN V[elem,cap]:=V[elem-1,cap]
         ELSE V[elem,cap]:= Max2(V[elem-1,cap],
                       (V[elem,cap-obj[elem].peso]+obj[elem].valor))
         END
      END
   END;
   RETURN V[i,p]
END Mochila3;
```

La complejidad del algoritmo es la que corresponde a la construcción de la tabla, es decir O(nM).

5.18 LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

Necesitamos calcular la matriz producto M de n matrices dadas $M = M_1M_2...M_n$ minimizando el número total de multiplicaciones escalares a realizar. Este problema ya fue planteado en el capítulo anterior, en donde vimos que los algoritmos ávidos presentados no encontraban solución en todos los casos.

Nos preguntamos ahora si existe un algoritmo que lo resuelva utilizando la Programación Dinámica.

Solución (☺)

En primer lugar, vamos a suponer como hacíamos en el capítulo anterior que cada M_i es de dimensión d_{i-1}xd_i ($1 \le i \le n$), y por tanto realizar la multiplicación $M_i M_{i+1}$ va a requerir un total de $d_{i-1}d_id_{i+1}$ operaciones. Llamaremos $M(i,j)$ al número mínimo de multiplicaciones escalares necesarias para el cómputo del producto de $M_i M_{i+1} ... M_j$ con $1 \le i \le j \le n$. Por consiguiente, la solución al problema planteado coincidirá con $M(1,n)$.

Para plantear la ecuación en recurrencia que define la solución, supongamos que asociamos las matrices de la siguiente manera:

$$(M_i M_{i+1} ... M_k) (M_{k+1} M_{k+2} ... M_j).$$

Aplicando el principio de óptimo, el valor de $M(i,j)$ será la suma del número de multiplicaciones escalares necesarias para calcular el producto de las matrices $M_i\ M_{i+1}...M_k$, que corresponde a $M(i,k)$, más el número de multiplicaciones escalares para el producto de las matrices $M_{k+1}M_{k+2}\ ...M_j$ que es $M(k+1,j)$, más el producto que corresponde a la última multiplicación entre las matrices de dimensiones
$(d_{i-1}\ d_k)$ y $(d_k\ \ d_j)$ es decir, $d_{i-1}d_kd_j$. En consecuencia, el valor de $M(i,j)$ para la asociación anteriormente expuesta viene dado por la expresión:

$$M(i,j) = M(i,k) + M(k+1,j) + d_{i-1}d_k\, d_j.$$

Pero k puede tomar cualquier valor entre i y j–1, y por tanto $M(i,j)$ deberá escoger el más favorable de entre todos ellos, es decir:

$$M(i,j) = \underset{i \le k < j}{Min}\{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\},$$

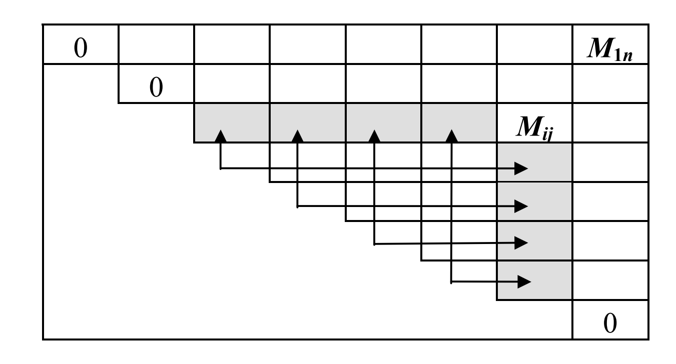
lo que nos lleva a la siguiente relación en recurrencia:

$$M(i,j) = \begin{cases} 0 & \text{si } i = j \\ \underset{i \le k < j}{Min}\{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\} & \text{en otro caso.} \end{cases} \qquad [5.4]$$

Para resolver tal ecuación en un tiempo de complejidad polinómico es necesario crear una tabla en la que se vayan almacenando los valores $M(i,j)$ $(1 \le i \le j \le n)$ y que permita a partir de las condiciones iniciales reutilizar los valores calculados en los pasos anteriores.

Esta tabla se irá rellenando por diagonales sabiendo que los elementos de la diagonal principal son todos cero. Cada elemento $M[i,j]$ será el valor mínimo de entre todos los pares $(M[i,k] + M[k+1,j])$ señalados con la línea de doble flecha en la siguiente figura, más la aportación correspondiente a la última multiplicación $(d_{i-1}d_kd_j)$. Los valores que requiere el cálculo de $M[i,j]$ y que el algoritmo reutiliza para conseguir una tiempo de ejecución aceptable se encuentran sombreados:

Al ir rellenando la tabla por diagonales se asegura que esta información $(M[i,k],M[k+1,j])$ está disponible cuando se necesita, pues cada $M[i,j]$ utiliza para su cálculo todos los elementos anteriores de su fila y todos los de su columna por debajo suya.



Rellenada la tabla, la solución la podemos encontrar en el extremo superior derecho, que nos indica el número de multiplicaciones escalares buscado, $M[1,n]$.

Si además queremos conocer cómo es la asociación que corresponde a este óptimo es necesario conservar para cada elemento $M[i,j]$ el valor de k para el cual la expresión $M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$ es mínima, construyendo otra tabla que denominamos *Factor*.

El siguiente algoritmo *Matriz* es de creación de las tablas *M* y *Factor*. En la tabla *M* se almacenan los valores del número mínimo de multiplicaciones y en la tabla *Factor* la información necesaria para construir la asociación óptima.

```
TYPE MATRIZ = ARRAY [1..n],[1..n] OF CARDINAL;
     ORDEN  = ARRAY [0..n] OF CARDINAL;(* dimensiones *)

PROCEDURE Matriz(VAR d:ORDEN;n:CARDINAL;VAR M,Factor:MATRIZ);
   VAR i,diagonal:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      M[i,i]:=0
   END;
   FOR diagonal:=1 TO n-1 DO
      FOR i:=1 TO n-diagonal DO
         M[i,i+diagonal]:=
               Minimo(d,M,i,i+diagonal,Factor[i,i+diagonal]);
      END
   END
END Matriz;
```

La función *Minimo* es la que calcula el mínimo de la expresión en recurrencia [5.4], y devuelve no sólo el valor de este mínimo, sino el valor de *k* para el que se alcanza (mediante el parámetro *k1*):

```
PROCEDURE Minimo(VAR d:ORDEN;VAR M:MATRIZ;i,j:CARDINAL;
                 VAR k1:CARDINAL):CARDINAL;
   VAR aux,k,min:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR k:=i TO j-1 DO
      aux:=M[i,k]+M[k+1,j]+d[i-1]*d[k]*d[j];
      IF aux<min THEN min:=aux; k1:=k END
   END;
   RETURN min
END Minimo;
```

Observando el procedimiento *Matriz* vemos que existe un bucle externo que se repite desde *diagonal* = 1 hasta $n - 1$, y en su interior un bucle dependiente de la iteración estudiada y del valor de *diagonal*, y que se ejecuta desde 1 hasta $n - diagonal$. En el interior de este bucle hay una llamada al procedimiento *Minimo* que tiene una complejidad del orden del valor de la *diagonal*, y por tanto el tiempo de ejecución del algoritmo es:

$$\sum_{diagonal=1}^{n-1}(n - diagonal)diagonal = n\sum_{diagonal=1}^{n-1}diagonal - \sum_{diagonal=1}^{n-1}diagonal^2 = (n^3 - n)/6$$

por lo que concluimos que su complejidad temporal es de orden $O(n^3)$. Por otro lado, la complejidad espacial del algoritmo es de orden $O(n^2)$.

En caso que deseemos reconstruir la solución a partir de la tabla *Factor*, el siguiente procedimiento muestra por pantalla la forma de multiplicar las matrices para obtener ese valor mínimo:

```
PROCEDURE EscribeOrden(VAR Factor:MATRIZ;i,j:CARDINAL);
   VAR k:CARDINAL;
BEGIN
   IF i=j THEN
      WrStr('M');
      WrCard(i,0)
   ELSE
      k:=Factor[i,j];
      WrStr('(');
      EscribeOrden(Factor,i,k);
      WrStr('*');
      EscribeOrden(Factor,k+1,j);
      WrStr(')')
   END
END EscribeOrden;
```

El algoritmo aquí presentado es capaz de encontrar el valor de la solución óptima junto con una de las formas de obtenerla. Sin embargo, existen casos en donde puede haber más de una forma de multiplicar las matrices para obtener el valor óptimo, como muestra el siguiente ejemplo.

Sean las matrices M_1 (10x10), M_2 (10x50) y M_3 (50x50). Existen dos formas de asociarlas para multiplicarlas, y en ambos casos obtenemos:

$$(M_1M_2)M_3 = 10\cdot10\cdot50 + 10\cdot50\cdot50 = 30000$$
$$M_1(M_2M_3) = 10\cdot50\cdot50 + 10\cdot10\cdot50 = 30000$$

Es posible modificar el algoritmo para que encuentre todas las soluciones que llevan al valor óptimo, y para esto es suficiente valerse de la matriz *Factor*, si bien esta modificación reviste poco interés desde un punto de vista práctico.